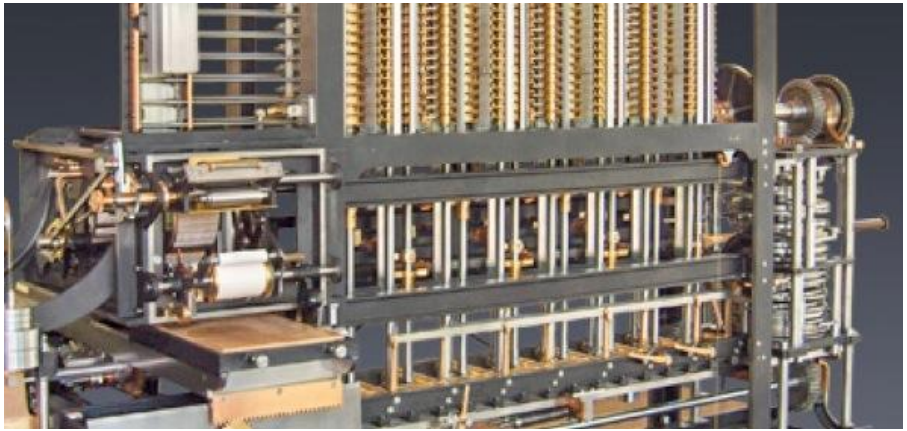
UE Programmation C++

Master Sciences pour l'Ingénieur

- Conversions de type
- Polymorphisme
  - Liens statiques
  - Fonctions virtuelles
  - Liens dynamiques
  - Classes abstraites



Cours 2

Alice Cohen-Hadria alice.cohenhadria@gmail.com
(Transparents B. Gas & ACH)

L'exemple de la classe Complexe


```
class Complex {
private:
      double re, im;
public:
      // Constructeur
      Complex( double re=0, double im=0 ) {
            cout << "je suis le constructeur par défaut" << endl;
            this->re = re;
            this->im = im;
      }

      // Constructeur par copie, appelé automatiquement si création d'une
nouvelle instance par copie
      Complex( const Complex &c ) {
            cout << "je suis le constructeur par copie" << endl;
            this->re = c.re;
            this->im = c.im;
      }
```

L'exemple de la classe Complexe

Rappel : Surcharge de l'opérateur =
A l'intérieur de la classe Complexe -> Doit retourner une référence sur Complexe

L'exemple de la classe Complexe

Rappel : Surcharge de l'opérateur =
A l'intérieur de la classe Complexe -> Doit retourner une référence sur Complexe

```
Complex &operator=( const Complex &c ) {
        if ( &c == this )
                return *this;

        cout << "je suis l'opérateur d'affectation" << endl;
        this->re = c.re;
        this->im = c.im;
        return *this;
}
```

Constructeur de conversion

Les attributs de classe re et im sont de type double.
On veut implémenter un constructeur qui fonctionnerait même si on ne passe pas le bon type en paramètre.

```
// Constructeur de conversion
Complex( int i ) {
        cout << "je suis le constructeur de conversion" << endl;
        this->re = double (i);
        this->im = 0.;
}
```

Le constructeur de conversion

Quizz :

https://app.wooclap.com/events/OCXHPF/0

Le constructeur de conversion

Réponse au quizz :

```
int main() {
      // Constructeur par défaut
      Complex C1;

      // Constructeur de conversion
      Complex C2 = 4;

      // Constructeur de conversion
      Complex C3 = Complex(4);

      // Constructeur par copie (pas l'opérateur d'affectation)
      Complex C4 = C1;

      // Opérateur d'affectation (pas le constructeur par copie)
      C1 = C2;
      C1 = C1;
}
```

Conversions de type : classe dérivée

Une Forme a un couleur.
Un Point est une sorte de Forme. Un Point possède deux attributs x et y, correspondant à ses coordonnées.

Conversions de type : classe dérivée

Une Forme a un couleur.
Un Point est une sorte de Forme. Un Point possède deux attributs x et y, correspondant à ses coordonnées.

```
class Forme {
   int couleur;

public:
   Forme(int c=0) : couleur(c){}
};

class Point : public Forme {
   int x, y;

public:
   Point(int c=0, int x=0, int y=0)
      : Forme(c), x(x), y(y) {}
};
```

Nouvelle syntaxe (empruntée au syntaxe de constructeur)
Équivalent à : `this.x = x;`

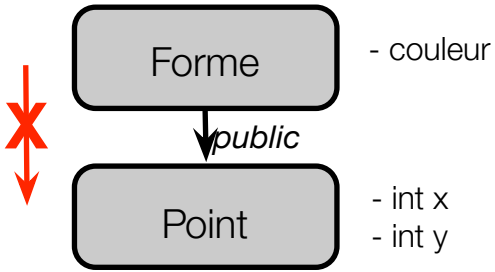
`int()` existe et vaut 0.
On peut aussi écrire: `int a (5);`

Conversions de type : classe dérivée

Une Forme a un couleur.
Un Point est une sorte de Forme. Un Point possède deux attributs x et y, correspondant à ses coordonnées.

La conversion d'une classe dérivée vers une classe de base privée ou protégée existe :

On ne peut pas convertir une Forme en Point

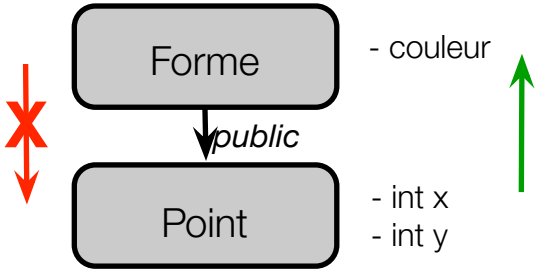


On peut convertir un Point en Forme

Conversions de type


```
Point p(3, 1, 2);
Forme f = p;  // EQV: Forme f = Forme(p);
Point p2 = f; // erreur !
```

```
error: conversion from 'Forme' to non-scalar
type 'Point' requested
```

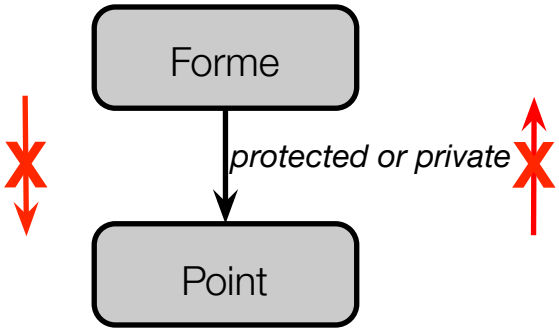


Tous les membres de la classe dérivée qui n'appartiennent pas à la classe de base sont ignorés.

L'inverse n'est pas possible.

Conversions de type

La conversion de type quand l'héritage est privé ou protected n'est pas possible

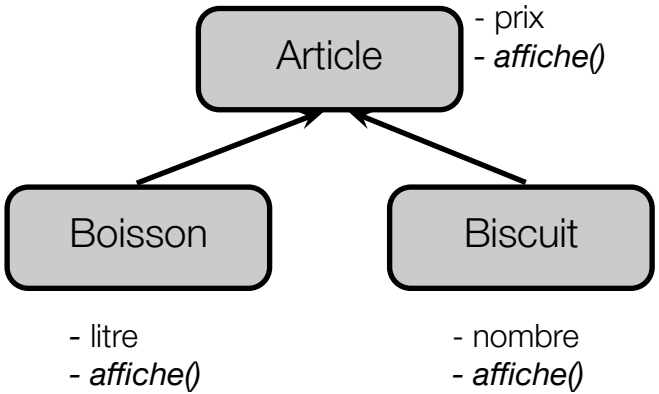



```
Forme f = p;
```

```
error: 'Forme' is an
inaccessible base of
'Point'
```

Liens statiques

On considère l'exemple suivant :



On a donc Boisson et Biscuit qui hérite de la classe Article.
Un article a un prix. Une boisson est en litre, un biscuit possède le nombre de biscuit dans la boîte.

Liens statiques


```
class Article
{
  protected:
    int prix;  // Chaque article a un prix
  public:
    void affiche() const { cout << "Article." << endl; }
};

class Boisson : public Article //Une Boisson EST UN Article
{
  protected:
    int litre ;  // Le volume de la boisson en litres

  public:
    void affiche() const {cout << "Boisson." << endl;}
};

class Biscuit : public Article  //Un Biscuit EST UN Article
{
  protected:
    int nombre;  // Le nombre de biscuits

  public:
    void affiche() const {cout << "Biscuit." << endl ;}
};
```

Polymorphisme d'héritage

- Avec la fonction principale :

```
int main(){
    Article a;
    a.affiche();

    Boisson b;
    b.affiche();

    Article c = b;
    c.affiche();
}
```

Article
Boisson
Article

affiche est présente dans les classes Articles et Boisson.
Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution.
On appelle cela l'**association tardive** (ou liaison tardive ou liaison dynamique).
Le **polymorphisme** représente la capacité du système à choisir dynamiquement la méthode qui correspond au type de l'objet en cours de manipulation.

Polymorphisme

Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution.
On appelle cela l'**association tardive** (ou liaison tardive ou liaison dynamique).
Le **polymorphisme** représente la capacité du système à choisir dynamiquement la méthode qui correspond au type de l'objet en cours de manipulation.

Buts du polymorphisme :

- Pouvoir choisir le point de vue le plus adapté selon les besoins
- Pouvoir traiter un ensemble de classe liées entre elles de **manière uniforme**
  - Exemple: liste d'`Article`

Liens statiques

- A présent on écrit la fonction suivante d'affichage :

```
void afficher( const Article &a)
{
    a.affiche();
}
```

- Le programme suivant affichera ?????

```
int main()
{
    Article a;
    afficher(a);

    Boisson b;
    afficher(b);

    return 0;
}
```

Liens statiques

- A présent on écrit la fonction suivante d'affichage :

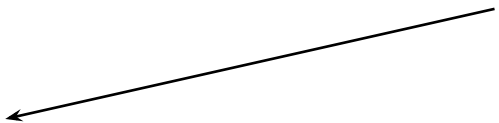
```
void afficher( const Article &a)
{
    a.affiche();
}
```

- Le programme suivant affichera : Article et Article

```
int main()
{
    Article a;
    afficher(a);

    Boisson b;
    afficher(b);

    return 0;
}
```



Liens statiques

La résolution du type du paramètre de afficher est faite à la compilation, et non pas à l'exécution comme pour les méthodes d'un objet.
On dit qu'elle est faite statiquement.

Lien statique : le compilateur génère un appel à un nom de fonction spécifique, et l'éditeur de liens résout cet appel à l'adresse absolue du code à exécuter.

Le paramètre est de type `Article` à la compilation.
À l'exécution, l'instance `b` est castée en `Article` pour satisfaire la signature de `afficher`

Liens dynamiques

On veut donner à la fonction afficher un comportement de polymorphisme.

```
void afficher( const Article &a)
{
    a.affiche();
}
```

`a.affiche()` soit résolu dynamiquement (à l'exécution) en fonction du type de `a`.

On déclare qu'on veut une fonction qui ait la flexibilité des propriétés de l'association tardive en utilisant le mot-clé `virtual`.

Fonctions virtuelles et liens dynamiques

- En déclarant les fonctions *virtual* :

```
class Article
{
   protected:
    int prix;  //Chaque article a un prix
   public:
    virtual void affiche() const { cout << "Article." << endl; };
};

class Boisson : public Article //Une Boisson EST UN Article
{
   protected:
    int litre ;  // Le volume de la boisson en litres
   public:
    virtual void affiche() const {cout << "Boisson." << endl;};
};

class Biscuit : public Article  //Un Biscuit EST UN Article
{
   protected:
    int nombre;  // Le nombre de biscuits
   public:
    virtual void affiche() const {cout << "Biscuit." << endl };
};
```

Liens dynamiques

`virtual` n'est pas obligatoire dans les classes filles.
Une fonction déclarée `virtual` dans une classe mère entraîne cette même propriété pour les fonctions redéfinies des classes filles.

Avec `virtual` :
- Liaison dynamique/tardive

Sans `virtual` :
- Liaison statique

Attention à la différence entre surcharge et redéfinition :
- Surcharge : signature différente
- Redéfinition : même signature d'une méthode dans une classe et sa classe fille.

Mot clé `override` pour éviter des bugs : intention pour la classe dérivée de remplacer le comportement d'une méthode virtual dans la classe de base.

Liens dynamiques

- Le code suivant :

```
void afficher( const Article &a){
    a.affiche();
}
```

- Avec la fonction principale :

```
int main(){
    Article a;
    afficher(a);    //Affiche "Article"

    Boisson b;
    afficher(b);    //Affiche "Boisson"

    return 0;
}
```

Liens dynamiques

Les classe qui implémentent des fonctions virtuelles disposent d'un pointeur supplémentaire (caché) dans chaque objet, et quelque soit le nombre de fonctions virtuelles.
Ce pointeur pointe vers une table qui contient les adresses qu'ont les fonctions virtuelles pour les objets de la classe en question.

Exemple :

```
#define maxArticles 6
int main(){

    Article *tab[maxArticles];
    int i=0;
    tab[i++] = new Boisson;
    tab[i++] = new Biscuit;
    ...
    for ( int j=0; j<2; j++ )
         tab[j]->affiche();

    return 0;
}
```

Si affiche n'est pas `virtual` dans Article :

????

Si affiche est `virtual` dans Article :

????

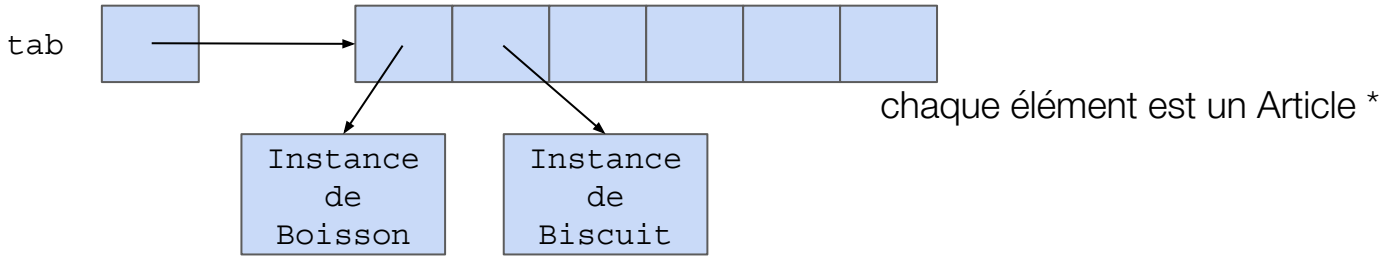
Liens dynamiques


```
#define maxArticles 6
int main(){

    Article **tab[maxArticles];
    int i=0;
    tab[i++] = new Boisson;
    tab[i++] = new Biscuit;
    ...
    for ( int j=0; j<2; j++ )
         tab[j]->affiche();
     return 0;
}
```

Si affiche n'est pas `virtual` dans Article : Article et Article

Si affiche est `virtual` dans Article : Boisson et Biscuit



chaque élément est un Article *

Liens dynamiques

La classe qui implémente une fonction virtuelle doit **impérativement déclarer un destructeur virtuel**.

Exemple :

```
class Article{
   protected:
    int prix;  //Chaque article a un prix


   public:
    virtual void affiche() const { cout << "Article." << endl; };

    virtual ~Article() {…}
};
```

Classes abstraites

Lorsque les fonctions virtuelles sont placées très haut dans la hiérarchie des classes, il se peut que leur implémentation n'ait plus vraiment de sens. Dans ce cas on définit des fonctions vides :

```
class Article{
    public:
    virtual void affiche() const { cout << "Article." << endl; };
    virtual int poids() {};

    protected:
    int prix;  // Chaque article a un prix
};
```

Classes abstraites

Il est possible, et recommandable, d'officialiser le fait qu'une fonction virtuelle est vide. Cela se fait en utilisant une fonction **virtuelle pure** :

```
class Article{
    public:
    virtual void affiche() const { cout << "Article." << endl; };
    virtual int poids() = 0;

    protected:
    int prix;  // Chaque article a un prix
};
```

Spécification d'un concept dont la réalisation diffère selon les sous classes :

- pas implémentée
- doit être redéfinie et implémentée dans les sous-classes instanciables

Classes abstraites

Une classe qui contient au moins une fonction virtuelle pure est une classe abstraite.

Il n'y a pas de création possible d'objets de ce type.

La déclaration d'un objet d'une classe abstraite génère une erreur du compilateur, ce qui n'est pas le cas lorsqu'on utilise des fonctions virtuelles vides.

Une classe abstraite qui n'a que des méthodes abstraites s'appelle une **interface.** (dans d'autres langages objets (ex java) interface est un mot clé).